DOCUMENT RESUME

ED 228 984 IR 010 652

AUTHOR Eisenstadt, Marc
TITLE Design Features of a Friendly Software Environment for Novice Programmers. Technical Report No. 3.
INSTITUTION Open Univ., Milton Keynes, (England). Human Cognition Research Lab.
PUB DATE Dec 82
NOTE 49p.
PUB TYPE Reports - General (140)

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS Artificial Intelligence; *Computer Programs; Databases; *Design Requirements; *Educational Environment; Foreign Countries; Higher Education; Instructional Materials; Man Machine Systems; Postsecondary Education; *Programing; Programing Languages; Psychology; *Teaching Methods
IDENTIFIERS *Authoring Languages; Open University (Great Britain)

ABSTRACT

This paper describes the results of a 6-year period of design, implementation, testing, and iterative redesign of a programming language, user aids, and curriculum materials for use by psychology students learning how to write simple computer programs. The SOLO language, which was the resulting product, is primarily a simple, database manipulation, program language, with an explicit "flow of control" construct in conditional statements which eliminates the elementary flow of control problems normally found in novices' programs. Following a description of project rationale and background, additional sections detail the programming language, the surrounding user environment, and the curriculum materials, listing the important principles underlying the design of each. Throughout the paper, the principles which could benefit designers of other software environments are emphasized. A 34-item reference list is included. (LMM)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED228984

THE OPEN UNIVERSITY

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☑ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official NIE position or policy.

DESIGN FEATURES OF A FRIENDLY SOFTWARE ENVIRONMENT FOR NOVICE PROGRAMMERS

Marc Eisenstadt

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY

Marc Eisenstadt

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

IR010652

Milton Keynes, MK7 6AA

HUMAN COGNITION RESEARCH LABORATORY

DESIGN FEATURES OF A FRIENDLY SOFTWARE ENVIRONMENT FOR NOVICE PROGRAMMERS

Marc Eisenstadt

Abstract: A highly friendly software environment for novice programmers, based on a non-numerical programming language called SOLO, has been used with demonstrable success by more than 2000 Social Science students. During the development of SOLO, a great deal of emphasis was placed on the interaction among three aspects of the overall design: the programming language itself, the user environment, and the curriculum used to introduce students to programming. The language has an 'explicit flow of control' construct in conditional statements which eliminates the elementary flow of control bugs normally found in novice's programs. The environment is simple, consistent, and 'visible' in that it displays all changes to the values of global variables as soon as they occur. The curriculum is non-standard: students begin with the concepts of 'assertional data base' and 'pattern-matching', which they find highly intuitive, motivating, and relevant to their academic interests. Features of the language, environment, and curriculum which could benefit other software designers are described in detail.

Key words and phrases: programming languages; programming environments; man-machine interface; human-computer interaction.

Acknowledgements: The author wishes to thank Vincent P. Cook (SOLO-78), Adam Gawronski (Micro-SOLO), Tony Hasemer (MacSOLO), Matthew Lewis (empirical studies of SOLO users), and Roger Moore (SOLO-82) for their constructive suggestions during the evolution of SOLO.

Human Cognition Research Laboratory
The Open University
Milton Keynes MK7 6AA
ENGLAND

Technical Report No. 3
December, 1982

1. INTRODUCTION

This paper describes the results of a six year period of design, implementation, testing, and iterative re-design of a programming language, user aids, and curriculum materials for use by Psychology students learning how to write simple computer programs. The resulting product, called 'SOLO'*, has now been used by more than 2000 Open University students. SOLO itself is a very simple programming language, but the design philosophy of the language, the user aids, and the curriculum materials embody many principles which could pay off if adopted in the teaching of other programming languages. The next section describes the background to the project and the reason it was undertaken. The remaining sections describe in turn the programming language, the surrounding user environment, and the curriculum materials, listing the important principles underlying the design of each of these. Throughout the paper, the principles which can benefit designers of other software environments are emphasized.

2. BACKGROUND

The Open University, in existence since 1969, is a 'long-distance' teaching institution [27]. We have 63,000

---

*The name SOLO is derived from a combination of SOL [25] and LOGO [26]. It is also meant to convey the image of Open University students who study at home on their own. The language SOLO has no connection with Project Solo [9] nor with the Solo operating system [5].

undergraduate students currently enrolled. Students typically hold full time jobs, studying at home for roughly 12 hours per week. We make extensive use of correspondence text, BBC television and radio programmes, small home experimental kits, and 1-week residential summer schools. Through a special arrangement with colleges and polytechnics around the country we have established 250 study centres where students can meet with tutors, and where they can use a computer terminal via dial-up facilities to one of our 3 DECsystem-20 computers. These computers run a variety of Computer Assisted Instruction packages, simulation games, and programming languages [4]. Students attending summer school have hands-on access to a large range of microcomputers, and an increasing number of Open University courses are simply mailing microcomputers directly to students as part of their home experimental kits.

In 1976 a team of Open University lecturers under the direction of Professor Judith Greene began to design a course on Cognitive Psychology. The course was to be presented for the first time in 1978. A course such as this runs for 32 weeks every year, for approximately 6-8 years, and is almost entirely pre-packaged, with the exception of yearly changes to some assignments and exam questions. Thus, there is tremendous pressure on us to get the material right before it goes into print. Two important principles were agreed by the course team early in 1976: (a) the course would contain a fairly large amount of material on

Artificial Intelligence (AI), since such material was becoming increasingly relevant to Cognitive Psychologists; (b) all the course materials would undergo two rounds of drafting and group criticism, after which a third draft of the materials would be produced and used by a pilot group of thirty students in 1977-- the comments of these 'developmental testers' would then provide the basis for revisions of the material to appear in the final curriculum in 1978.

Our desire to include a lot of AI material meant that our students had to become conversant with a number of basic ideas involving computation. We felt this was essential, because it would enable our students to feel more comfortable when encountering AI articles or recent Cognitive Science theories expressed in some computational form. The best way to achieve this was clearly to require the students to undertake an elementary programming project. The goals of the project are summarized in this excerpt from the students' project booklet [10]:

> "By the time you finish this project you should be able to:
>
> 1) connect to and log in to the computer
>
> 2) define the following terms, and state their relevance to models of cognitive processing:

symbol	relation
computer	pattern-matching
program	parameter
programming language	variable
procedure	flow of control
subroutine	recursion
data base	iteration
semantic network	inference
node	proposition

3) write working computer programs using all of the above concepts;

4) explain the aspects of cognitive processing which are embodied in your programs;

5) explain why 'representation' is a problem."

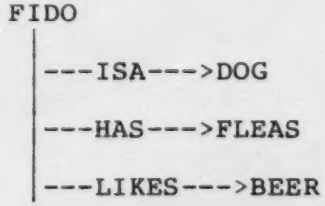
The Open University, as one of its founding principles, has no entrance requirements, and our course has no prerequisites. Thus, we had to proceed with great caution--we had to produce a non-threatening experience of computer programming for students who would be working alone at remote study centres, who had no prior computing experience, and who were not interested in computing in the first place.

Our results have been very encouraging. If nothing else, we have succeeded in giving the vast majority of our students a pleasurable encounter with computers in potentially intimidating conditions. A majority of students coming to our residential summer schools consistently choose a programming project as their first choice out of four projects-- and this is with the selection process weighted against the programming project, because those few students (10%-20%) who have prior programming experience are advised to choose one of the other projects. How, then, did this state of affairs come about? The short answer is that we

designed a programming language which enabled students to do powerful things on the first day, embedded this language in a friendly software environment which corrected obvious errors (such as silly spelling mistakes) automatically, and developed a curriculum sequence which from start to finish tried to motivate the student by highlighting the relevance of each programming task to the student's main academic interest-- cognitive psychology. One of the main messages of this paper will be that the language, the environment, and the curriculum all warrant equal attention from would-be software designers. The following sections present a detailed discussion of each.

3. THE LANGUAGE

SOLO is primarily a data base manipulation language. User-defined procedures can invoke primitives which add, remove, retrieve, or print out structures in a relational data base. The data base is a unidirectional labelled graph, with arbitrarily-named nodes connected by arbitrarily-named relations to other nodes, as in the following structure:



Manipulation of this relational network is accomplished by means of the following ten primitives:

(1) NOTE: allows relational 'triples' to be added to the data base, e.g.

```
NOTE FIDO---EATS--->MEAT
NOTE FELIX---ISA--->CAT
```

(2) FORGET: allows relational triples to be removed from the data base, e.g.

```
FORGET FIDO---HAS--->FLEAS
```

(3) CHECK: allows the user to test for the presence or absence of relational triples, e.g.

```
CHECK FIDO---LIKES--->BEER
```

would return 'present', given the first description of the node FIDO shown above, whereas

```
CHECK FIDO---LIKES--->WINE
```

would return 'absent'. This present/absent dichotomy (rather than true/false) is used as the basis for conditional branching in user-defined procedures. CHECK also allows the use of 'wild cards' in the final position of the relational triple. For instance, given the above description of FIDO,

```
CHECK FIDO---LIKES--->?X
```

would return 'present', and would also assign to X the value "BEER". SOLO uses an implicit 'reverse quoting' convention comparable to that used several AI languages (e.g. CONNIVER [23]), so that the node whose name is 'FIDO' is simply referred to by the four character string FIDO. Conversely, the value of the variable X is referred to be prefixing it with an asterisk, i.e. *X. Our students find this

pattern-matching style of memory retrieval, the emphasis on assertional data bases, and the use of 'reverse quoting' all highly intuitive, and they have no problems performing simple pattern-matching retrieval from the data base during their first terminal session.

(4) FOR EACH: allows the user to iterate through sets of common relation-names emanating from a given node, e.g.

```
TO ANALYZE /X/
10 FOR EACH CASE OF /X/---DRINKS--->?Y
  10A PRINT "HERE IS ANOTHER ONE:" *Y
DONE
```

As with CHECK, sub-line numbering is supplied automatically by SOLO.

(5) PRINT: allows the printing of arbitrary messages, in the form of quoted strings, interspersed with variables if desired, e.g.

```
PRINT "I SEE THAT FIDO LIKES TO DRINK" *X

PRINT "I SEE THAT" *X "IS THE THING FIDO LIKES"
```

(6) TO: allows the user to define his or her own procedures. Parameters may be used, indicated by enclosing them in slashes. Line numbering is similar to that used by BASIC and LOGO, e.g.

```
TO PRAISE /X/
10 NOTE /X/---IS--->MARVELLOUS
20 PRINT "I THINK THAT" /X/ "IS REALLY GREAT"
DONE
```

Automatic line numbering is available, as is screen-oriented editing without any line numbering. These features are discussed as 'environment' issues in section 4.

The design of the language encourages a top-down, modular style of programming. Procedures may have at most ten main steps. No GOTOs are allowed. The present/absent dichotomy, described above, provides a standard if-then-else style of conditional with the following three restrictions: a) no nesting of conditionals is allowed; b) only one action (sub-procedure or primitive invocation) can be performed on each branch of the conditional; c) the user must specify whether the interpreter is to CONTINUE on to the next main step or EXIT back to the calling procedure. Sub-line numbering, indenting, and 'present/absent' prompting (illustrated *in italics* below) are all provided automatically by SOLO, e.g.

```
TO ASSESS /X/
10 CHECK /X/---VOTES--->LIBERAL
  10A If present: PRAISE /X/; CONTINUE
  10B If  absent: CONDEMN /X/; CONTINUE
20 PRINT "I'M NOW DONE ASSESSING" /X/
DONE
```

In this case, the user may have already defined PRAISE, as in the preceding example, but can wait to define CONDEMN later on.

(7) EDIT: allows the user to retype old lines or insert new ones within the definition of any existing procedure. Essentially EDIT mode and DEFINE mode ('TO' mode) are identical. Lines are inserted or overwritten just by typing in the line number followed by the desired SOLO statement. A line number followed by a carriage-return deletes that particular line. EDIT mode and DEFINE mode are both terminated by typing the word DONE. A screen editor,

based on the syntax-directed editing philosophy of the Cornell Program Synthesizer [31] is available to summer school students using our microcomputer implementation of SOLO (described below).

(8) LIST: prints out the definitions of student-defined procedures. The example below illustrates a simple recursive procedure intended to capture the inference that 'someone who gets the flu passes it on to whomever he or she kisses, and the new person passes it on in the same manner':

```
LIST INFECT

TO INFECT /X/
10 NOTE /X/---GETS--->FLU
20 CHECK /X/---KISSES--->?Y
  20A If present: INFECT *Y; EXIT
  20B If  absent: EXIT
DONE
```

In the above example, the symbol '*Y' (line 20A) is a backward reference to the 'wild-card' pattern-match variable '?Y' (line 20). The recursive invocation of INFECT on line 20A thus causes the main effect (NOTE /X/---GETS--->FLU) to propogate along a chain of KISSES relations.

(9) DESCRIBE: prints out the data base structures emanating from a given node, e.g.

```
DESCRIBE FIDO

FIDO
  |
  |---ISA--->DOG
  |
  |---HAS--->FLEAS
  |
  |---LIKES--->BEER
```

(10) BYE: allows the user to log off, and automatically saves the entire data base and all user-defined procedures under a unique identifier for each SOLO user.

This completes the description of the SOLO kernel. The kernel has remained unchanged over the years, although some extra facilities have been added in order to give our students improved tools in the light of empirical studies [12, 21]. There have been four main implementations of SOLO to date:

a) SOLO-78: This comprised the kernel described above, plus an extensive HELP feature, illustrated in section 4. SOLO-78 was used by over 1600 students between 1978 and 1981.

b) SOLO-82: This is an omptimized version, running several times faster than SOLO-78, and in current use by our students (400 users so far). Its HELP facility has been streamlined to take account of those HELP features not accessed by SOLO-78 users. In addition to the kernel, this version has a BASIC-style INPUT primitive, plus a more compact syntax for searching through several successive links in the data base. SOLO-82 users can type in

```
LET *B = FIDO'S MOTHER'S BROTHER
```

instead of the more cumbersome SOLO-78 sequence shown below:

```
CHECK FIDO---MOTHER--->?M
 If present: CONTINUE
 If  absent: EXIT
CHECK *M---BROTHER--->?B
 If present: CONTINUE
 If  absent: EXIT
etc.
```

A small number of extra utilities are provided for simplifying editing and bookkeeping. For example, KILL gets rid of unwanted procedures.

c) Micro-SOLO: This UCSD-Pascal implementation of SOLO has all of the features of SOLO-82, plus a simple screen editor which displays syntactic 'templates' to the user in order to minimize input errors [11]. It has been used by over 200 of our students at summer schools.

d) MacSOLO: This is our experimental MacLISP implementation, containing a smarter front-end, and a tutorial package incorporating automated program-understanding and debugging aids [15, 19, 20]. It has been used by 40 students, all volunteer experimental subjects.

The word 'SOLO' will be used throughout this paper to refer to the kernel of the language shared by all four dialects, with exceptions pointed out as appropriate.

In describing the kernel of SOLO, the only aspect of the environment which has been discussed so far is the automatic numbering of sub-lines within conditionals and FOR EACH loops. The environment is very important, of course, but discussion of it in this section has been minimized in

order to highlight those aspects of the language itself which are crucial. Here are the design features of SOLO which have yielded high rewards for its users:

(1) Simple virtual machine. The abstract 'SOLO machine' is extremely simple: it is a 'data base machine' consisting of two halves - a data base, and a pool of procedures. This is precisely how it is presented to students, as shown in Figure 1.

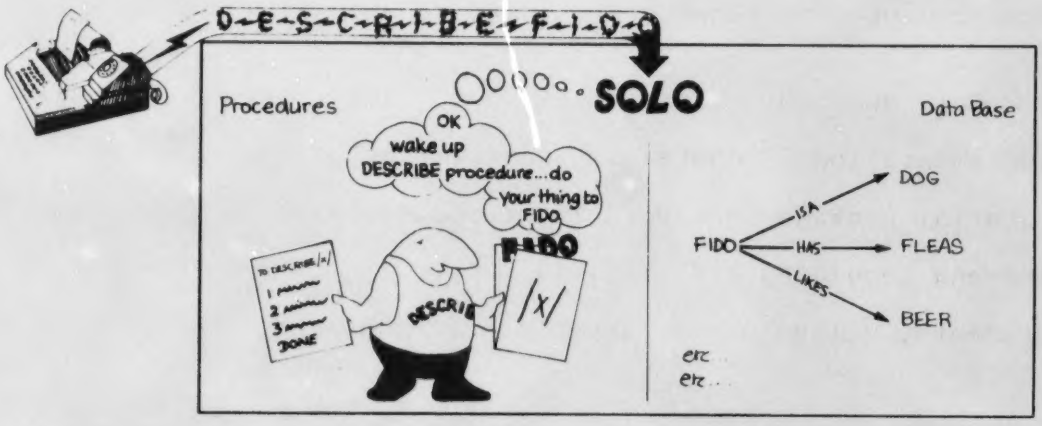


Figure 1. The SOLO 'virtual machine', as presented to students (reprinted from [10] with permission)

Procedures are described as 'autonomous experts' which know how to do things to the data base. The pool of procedures (NOTE, etc) can be augmented by user-defined procedures, which are simply new bundles of autonomous expertise. Subroutine calls are described to the student as

'activations' of procedures. Activation from the top-level (e.g. typing in NOTE FIDO---ISA--->DOG) is conceptually identical to sub-procedure activation (e.g. having step 10 of your own procedure perform NOTE FIDO---ISA--->DOG).

In terms of the 'levels of programming knowledge' described by Mayer [22], SOLO's clarity stems from the fact that there is a one-to-one mapping between the 'statement level' and the 'transaction level'. For example, the statement

NOTE FIDO---ISA--->DOG

corresponds to the following Mayer-style transaction:

Operation: AddToDataBase (= NOTE)
Object: [FIDO ISA DOG]
Location: DataBase

The importance of a simple virtual machine for teaching purposes has been nicely described by du Boulay, O'Shea and Monk [8]. Their work influenced the design of SOLO, and this principle appears to have been very beneficial to our students.

(2) Consistent use of virtual machine. The idea of consistency, also emphasized by du Boulay et. al., is particularly reflected in the wording of error messages. For example, suppose that a student simply typed in:

FIDO ISA DOG

Ignoring for the moment clever user-aids which can infer that the word NOTE was omitted (MacSOLO does this), what sort of error message should the student be presented with?

Possibilities include such things as:

SYNTAX ERROR

or

I DON'T UNDERSTAND 'FIDO' IN THIS CONTEXT

However, students are told that SOLO works by applying a procedure to arguments, and that the first word on a line is always the name of a procedure. Thus, the actual (and most consistent) error message is the following:

I DON'T HAVE A PROCEDURE CALLED 'FIDO'

This consistency pays tremendous dividends, as it helps pinpoint to the students exactly what has gone wrong and why. Consistency trades off against the convenience which could be provided by a 'Do what I mean' facility [32]. This issue is discussed further in Section 6.

(3) Powerful mnemonics. The English meanings of the primitive names NOTE, FORGET, DESCRIBE, etc., correspond very closely to the actual jobs they perform within the SOLO virtual machine. This makes them easy to remember. Good mnemonics, however, do not require a software designer to develop an 'English-like' programming language, as the next design principle emphasizes.

(4) Not 'English-like'. A pilot version of SOLO in 1976 was modelled closely on SOL [25] and included syntactic constructs such as the following:

CONNECT "FIDO" TO "DOG" WITH "ISA"

This was abandoned in favor of NOTE FIDO---ISA--->DOG after a brief trial run in which it became apparent that

a) all of the extra keystrokes were an enormous burden for beginners

b) pilot students found the syntax harder to remember, even though it was English-- they were tempted to try closely related constructs such as:

LINK "FIDO" UPTO "DOG" VIA "ISA"

c) pilot students imparted too much intelligence to other aspects of the SOLO machine, assuming it could understand English.

In addition, some pilot students said that they actually felt more secure about the activity of programming if it was kept deliberately distinct from the activity of communication in English. This distinctness also applied to the use of symbols such as * (which refers back to the result of the pattern-match involving the symbol ?). Other symbols, such as the word IT, could have been used to refer to the result of a pattern-match, but our pilot students actually preferred a special symbol. Although the course team agonized over what symbol to use, our pilot students said in the end that they didn't care what symbol it was, as long it was unique and easy to detect in their programs.

(5) Iconic symbols. The data base descriptions contain 'arrows' which look nearly like the descriptions one would draw on a blackboard, e.g.

NOTE FIDO---ISA--->DOG

The arrows may in fact be omitted by the user on input. SOLO-82 supplies them as soon as the space bar is pressed.

Micro-SOLO displays them on the screen in advance, as part of a template to be filled in. SOLO-78 and MacSOLO supply the arrows when procedures are LISTed.

The symbol which signifies parameters is a slash, e.g. /X/. This symbol helps to remind the students of the 'slot'-like role of parameters. The analogy of 'slot' is used extensively in the primer for the language (Section 5 discusses other issues specific to the SOLO curriculum).

(6) Distinct symbols for distinct usage. The five symbol groupings X, "X", /X/, ?X and *X have the following respective meanings:

X can only be the name of a node.

"X" can only be a character string, and it can only appear as an argument of PRINT.

/X/ can only be a formal parameter of a user-defined procedure.

?X can only be an unbound variable awaiting a successful pattern-match.

*X can only be a variable bound as the result of a successful pattern-match.

These distinctions satisfy the criterion of consistency mentioned earlier, but may turn out to be inconvenient for students who type the wrong symbol by mistake. Since the surrounding context is often sufficient to determine a student's true intention, it is up to the environment (user-aids) to cope with such mundane lexical and syntactic mistakes. From the standpoint of the language alone, the

use of these distinct conventions helps to clarify the different roles that X may play in different contexts.

(7) Entirely non-numerical. The language is geared towards data-base manipulation-- this is not an add-on feature. This is particularly useful for those students who feel uneasy about their mathematical skills.

(8) Small number of primitives. SOLO-78 has only ten primitives, yet these primitives provide enough power for beginners to do interesting projects. The small number of primitives reduces the cognitive load for the beginner. For those who still have trouble remembering, the entire language is neatly summarized on the back cover of the primer.

(9) Uniform syntax. The four 'workhorse' primitives (NOTE, FORGET, CHECK, FOR EACH) share the same syntax (i.e. node---relation--->node). Of the others, only TO has an idiosyncratic syntax, i.e.

TO procedure-name (optional parameters)

(10) Encourages top-down programming style. Since SOLO is interpreted, special forward declarations are not necessary for procedures which have yet to be defined. The student is encouraged simply to put in the names of such procedures when they are needed, then define them later.

(11) Forced modularity. By restricting SOLO-78 procedures to ten main steps, students are forced to break problems into manageable sub-problems, and to define sub-procedures to deal with each of the sub-problems. A similar restriction was used successfully by Brown & Rubinstein [6], who were teaching LOGO to humanities undergraduates.

The number ten is somewhat arbitrary, however, and students can have trouble keeping track of the hierarchy of subroutine calls which result from this restriction. Nevertheless, the restriction pays high dividends in terms of the modulatiry and coherence of the resulting programs. Micro-SOLO users are encouraged, but not forced, to keep procedure lengths down to less than one screenful (which turns out to be twenty lines).

Modularity is also imposed by allowing students only one action on each branch of a conditional - this must be a subroutine call if more than one action is desired.

(12) No backward jumps allowed. SOLO imposes a weak form of structuring on students' programs by encouraging the use of FOR EACH loops and outlawing general GOTOs. Some of our students (those who have used BASIC) complain about this, but most of them find the flow of control very easy to specify. The one exception to this is the difficulty of exiting a loop after precisely n iterations. This is only difficult because SOLO deliberately has no numerical

primitives. Students can add their own 'count' nodes into the data base, but this is cumbersome, so we are considering adding counting primitives for the students' benefit.

A restricted form of GOTO is allowed by means of the CONTINUE control-statement on conditional branches. This feature is discussed under point 14 below.

(13) Imperative-style conditionals. We adopted an imperative-style construction for conditionals using

CHECK node---relation--->node

instead of

IF PRESENT (node---relation--->node)

This is analogous to the use of TEST in some dialects of LOGO, e.g. TEST A = B rather than IF A = B. The virtue of this style is that it allows beginners to experiment with conditional tests at the top level of the interpreter. Students can just type in such tests and have the results ('PRESENT' or 'ABSENT') immediately displayed. An additional bonus for SOLO users is that the syntax for CHECK is the same as that for NOTE and FORGET.

When students begin inserting conditional clauses into their own procedures, they can continue using the same construct, but now they get prompted directly for the extra lines which constitute the 'then' and 'else' branches of the conditional clause (in SOLO these are called 'if present' and 'if absent'). The fact that the prompting and indenting are done automatically is obviously an environment feature

rather than a language feature, but the point here is that it is the language feature (imperative style) which makes the environment feature (automatic prompting and indenting) possible, and thus minimizes the burden of students having to remember a special syntax for conditional clauses.

(14) Obligatory 'then' and 'else' branches on conditional clauses. Students must specify what action is to be taken on each branch of a conditional (this includes 'no action'). This forces students to be clear about the intended effects of their procedures, since they must always think about the case they weren't catering for (e.g. 'what should I do just in case that relational triple is absent from the data base, unlikely as that may be?').

This feature, combined with those described in points 15 and 16 below, makes the flow of control in student-defined procedures exceptionally clear.

(15) Annotated conditional branches. The labelling of SOLO's conditional clauses with 'if present' and 'if absent' provides an unambiguous annotation which greatly simplifies the task of program design and debugging. The provision of an (albeit simple) annotation was influenced by the work of Sime, Green, and Guest [29], who found that explicitly labelled conditional clauses, such as the one shown below, were easier to write, easier to understand, and less error-prone than Algol-style conditional clauses.

```
IF LEAFY
  THEN BOIL
NOT LEAFY
  FRY
END LEAFY
```

In the above example 'NOT LEAFY' (like SOLO's 'if absent') is much more explicit than 'else' would be, as it reminds the programmer of precisely the circumstance in which 'FRY' will occur.

(16) Obligatory flow-of-control specification on conditional branches. After specifying some action (or no action) to occur on each branch of a conditional clause, students must add flow-of-control information in the form of 'EXIT' (analogous to 'RETURN' in FORTRAN or BASIC) or 'CONTINUE' (which means continue with the next main step). This forces students to think clearly about what should be happening next in their programs, and helps them to avoid the classic 'falling through conditional' bug which plagues most beginners. This construct is to some extent an artifact of our desire to stick to line-at-a-time procedure entry and editing. Yet in the hundreds of student programs we have looked at [12, 21], we have never seen an example of the standard beginner's flow-of-control error illustrated by the BASIC and LOGO programs below (line 20 is always executed, but the intent is to have it executed only when A≠B).

```
BASIC:                              LOGO:
 10 IF A=B THEN PRINT "YAY"          10 TEST A=B
 20 PRINT "BOO"                      15 IFTRUE PRINT "YAY"
                                     20 PRINT "BOO"
```

With the exception of SOLO-78, all versions of SOLO now have

primitive tests for equality which use automatic prompting and indenting conventions analogous to those of CHECK. Thus, the above BASIC and LOGO examples would look like this in SOLO-82:

```
10 TEST *A = *B
 10A If yes: PRINT "YAY"; EXIT
 10B If  no: PRINT "BOO"; EXIT
```

4. THE ENVIRONMENT

In order to make SOLO a relatively painless experience for novices, a great deal of effort was put into the total software environment within which these novices would be working. The important features of this environment are most easily illustrated by example. The example below is contrived to illustrate a large number of features in a short space, but is otherwise authentic, in that it will run as shown using SOLO-78. The example depicts a student logging in and adding a few nodes and relations to the data base, then trying to define the procedure ASSESS which was illustrated earlier during the discussion of the SOLO primitive 'TO'. Student input is shown **IN BOLD LIKE THIS.** The numbers at the left refer to the environment features which are discussed following the example. As you read through the example, keep in mind the profile of the Open University students taking the course, as described in Section 2. These are students who may never have used a typewriter before.

(1) **@LOGIN D303 AI 1010**

Good evening.
Welcome to SOLO.

(2) Student Number Please: **C0123865**

Messages

Type
HELP
for advice. Be sure to press the 'RETURN' key after every line you type.

(3) Do you want a typing lesson? (YES or NO): **NO**

SOLO: **NOTE JOHN ISA PERSON**

OK...

(4)
```
JOHN
  |
  |---ISA--->PERSON
```

SOLO: **NTE JOHN VOTES LIBERAL**

(5) (NTE = NOTE)

OK...

```
JOHN
  |
  |---ISA--->PERSON
  |
  |---VOTES--->LIBERAL
```

SOLO: **TO ASSESS /X/**

....: **10CHECK /X/ VOTES LIBRAL**

(10CHECK = 10 CHECK)

(6) When you typed 'LIBRAL' did you mean 'LIBERAL'?
(YES or NO): **Y**

(7) ..... 10A If present: **PRAISE /X/**

(8) (Please specify EXIT or CONTINUE after the ';' below)

```
..... 10A If present: PRAISE /X/; EXIT

..... 10B If absent: CONDEMN /X/; CONTINUE

.....: 10A

(9)  ..... 10A If present: PRAISE /X/; CONTINUE

.....: 20 PRINT THAT'S ALL I HAVE TO SAY ABOUT" /X/

(Left hand quotes assumed)

.....: BYE

(10) (BYE interpreted as DONE)

OK... I now know how to 'ASSESS' /X/

SOLO:  LIST ASSESS

(11) TO ASSESS /X/
     10 CHECK /X/---VOTES--->LIBERAL
       10A If present: PRAISE /X/; CONTINUE
       10B If  absent: CONDEMN /X/; CONTINUE
     20 PRINT "THAT'S ALL I HAVE TO SAY ABOUT" /X/
     DONE

     SOLO: ASSESS MARY

     Oops... I don't have a procedure called 'CONDEMN'

(12) SOLO:  HELP

     You've tried to use a procedure without first
     having defined it. See pp. 37-39 of Units 3-4
     for how to do this, or else type
       HELP PROCEDURES
     By the way, this error may simply have been
     caused by a spelling mistake!

(13) SOLO: SOS

     (Tutor being informed, please wait)...

     [LINK from EISENSTADT (Advising)]

     (talk): Hi, what's the problem?

     (talk): I'M WORKING ON THE 'ASSESS' EXAMPLE, AND
             CAN'T FIGURE OUT WHAT THE DEFINITION OF
             'CONDEMN' SHOULD BE.

     (talk): You're free to define your own version. If you
             would feel happier modifying an existing
             version, then you can copy mine, and edit it
```

later. When you see the SOLO: prompt, type in
COPY CONDEMN TUTORME1. Then you can
LIST CONDEMN or EDIT CONDEMN, just
like any other procedure.

(talk): **OK, THANKS**

(talk): See you

[Advice terminated]

(14) SOLO: **COPY CONDEMN TUTORME1**

SOLO: **LIST CONDEMN**

```
TO CONDEMN /X/
10 PRINT /X/ "IS EVIL"
20 PRINT "EVERYONE DISLIKES" /X/
DONE
```

SOLO: **EDIT CONDEMN**

....: 20 NOTE /X/ THINKS FUNNY

....: DONE

OK... I have re-defined how to 'CONDEMN' /X/

(15) SOLO: **HELP OPTIONS**

Hold down the 'CTRL' key and press the letter 'O' to suppress the detailed description of available options.

Here are the options you are allowed to request (a '+' below means that this option is in effect by 'default' when you initially log in to SOLO):

(Options related to spelling correction):

SPELL (+) Enables any misspellings of procedure names to be automatically corrected.

NOSPELL (-) Completely disables spelling corrector, thereby eliminating the next 4 options.

TRUSTING (+) Performs most corrections without asking approval.

{student presses CTRL-O to suppress message}
^O...

Type option here: **TRACE**

```
     Do you want to select any more options
     (Yes or No): NO

(16) SOLO:  ASSESS MARY

      (ASSESS)
        10
        10B

         (CONDEMN)
           10

      MARY IS EVIL

           20

            OK...

             MARY
               |
               |---THINKS---FUNNY

        20

      THAT'S ALL I HAVE TO SAY ABOUT MARY

     SOLO:  BYE

(17) Goodbye
```

Features of the SOLO environment which should be of interest to designers of software environments for novice programmers are listed below. The numbers correspond to the numbers on the left hand side of the sample interaction presented above.

(1) Login trap. Timesharing users are automatically connected to SOLO upon logging in, to minimize their interactions with other systems/monitors which are totally irrelevant to their coursework. This gives them the illusion of interacting entirely with a SOLO machine, which is consistent with the way SOLO is taught. Micro-SOLO

achieves the same effect through an auto-start bootstrap facility which takes effect as soon as the power is switched on.

(2) Personalized environment. We maintain a separate data base for each student. This helps to reinforce the idea of a personalized SOLO machine, and encourages experimentation with changes to the data base. From the user's point of view, SOLO has an infinitely large memory, and all procedure definitions and changes to the data base are permanently stored (automatically).

(3) Typing lesson for first time users. Interaction via a QWERTY keyboard can be a daunting task in itself, so SOLO-78 provides a separate typing lesson for our novices. In the example shown, the user chooses to bypass the lesson. Both the typing lesson and this (possibly annoying) question are removed after the first few sessions.

(4) Visible workings of inner machine. The effect of any change to the data base is displayed to the user the instant that it occurs. This minimizes the possibility of hidden surprises. Experienced users can easily disable this capability.

(5) Automatic spelling correction of obvious errors. 'Obvious' refers to near-misses of procedure names, since any given student defines very few procedures. The spelling correction algorithm is a slightly modified version of the algorithm outlined in [32].

(6) User confirmation of tricky spelling cases. Node names and relation names are considered 'tricky', since users can have hundreds of them, and many of them differ by only one or two characters. Any near-miss results in a request for confirmation from the user.

(7) Automatic prompting and indenting for both branches of conditional clauses. The rationale behind the conditional clause construct was discussed in Section 3 as a language feature. The important aspect of the user environment in this example is the way in which it forces the student to specify some action on each branch of the conditional. It also eliminates the need for the user to remember the idiosyncratic syntax of conditional clauses. In terms of Mayer's 'levels of programming knowledge' [22], the environment automatically reminds the user about the language's 'mandatory chunks'. This feature provides an elementary syntax-directed editing capability (cf. [31]) for users with access only to hard-copy terminals.

(8) Forced explicitness with respect to flow of control. This is an example of a language feature which is handled in a smooth way by the user interface. The user is simply reminded of the requirement, and re-prompted with the relevant line (with the semicolon supplied).

(9) Inviolability of complex syntax by later editing. In this example, the user has completely eliminated line 10A. Since both branches of the conditional clause are mandatory, the user is instantly re-prompted for the 'then' branch of the conditional, as shown. Thus, there is no way to destroy the integrity of a multi-line clause such as 10-10A-10B, even by editing the three lines separately. This principle extends to such things as students typing in "10A If absent", which is detected as an anomaly, since all 'A' sub-lines must be 'if present' branches. The Micro-SOLO screen editor imposes similar constraints, prohibiting the deletion of obligatory sub-lines.

(10) Interpretation of appropriate intentions, given the surrounding context. This capability is very difficult to achieve in general. There are many cases, however, where the current state of the machine provides strong constraints on what the user must have meant. In this example, 'BYE' only makes sense in the context of a top-level 'SOLO' prompt. Given an edit-level '...:' prompt, 'DONE' is a more natural interpretation, so this interpretation is assumed.

(11) Normalized output (prettyprinting). SOLO-78 and MacSOLO provide a normalized version of the node---relation--->node 'arrows' (upon output) to help reinforce the idea of a relational data base, and to make the user's code easier to read. The user is not, however, required to include these arrows on input. In SOLO-82 and Micro-SOLO the arrows are supplied automatically during user

input, so that 'what you see is what you get'.

(12) HELP facilities. If a user types HELP immediately following an error message, an expanded version of that message appears. If there is no immediately preceding error message, a general index of available HELP facilities is printed. HELP followed by a keyword gives specialized advice related to that keyword. The keyword OPTIONS is explained below (point 15).

(13) On-Line (human) tutorial aid. We experimented with a student-tutor teleconferencing facility (using the DECsystem-20's LINK and ADVISE capabilities). If students typed 'SOS', then a bell rang on the terminal of one of the tutors (provided that an appropriately designated tutor was logged in). This facility was only available in SOLO-78. Some students found it valuable, although communicating in this fashion was somewhat slow. It was mainly useful as a source of data about tutorial dialogues. Interestingly, when students were given the choice of explaining their problems in English versus demonstrating them in SOLO, they typically chose English, but then had to revert to demonstrations (at the tutor's request) when they had difficulty articulating their problems clearly in English. One conclusion this experience leads to is that debugging assistants and programmers' apprentices (e.g. [28, 30, 33]) should not attempt to process natural language input from the user, especially if the user is a novice.

(14) Ability to share code with other students. SOLO-78 users could copy code from other students or tutors at summer school. Students appreciated the ways in which copying could enhance group problem solving activities, and did not worry about the security or privacy of their programs. Some students used this capability to devise an 'underground' mailbox utility.

(15) Ability for users to modify the environment. SOLO-78 users were able to select from a variety of 'options', which altered certain aspects of the environment. For instance, students could suppress the 'visible machine' principle, by selecting the option 'SILENCE'; they could switch off the spelling corrector by typing 'NOSPELL', and so on. These options were embedded within a separate facility (the 'HELP OPTIONS' facility) rather than made directly accessible within SOLO because options like SILENCE do something fundamentally different from SOLO primitives like NOTE and CHECK. Unfortunately, the HELP OPTIONS interaction of SOLO-78 was clumsy. For one thing, an investigation of transcripts of a large sample of user interactions [12, 21] revealed that some of the options, e.g. TRUSTING and CAUTIOUS (which fine-tuned the spelling corrector) were never used. Following suggestions made by Lewis [21], later dialects of SOLO use the primitives ENABLE and DISABLE, so that students can type in ENABLE TRACE or DISABLE SPELL, etc. The keywords ENABLE and DISABLE still demarcate the special nature of the TRACE and SPELL

facilities to which they are applied, but are conveniently accessible from within SOLO.

(16) Tracing of user-defined procedures. Tracing facilities are important debugging tools (see [24] for a good historical overview). SOLO-78's tracing facility just shows the name of each procedure as it is entered, and each step number as it is executed. Sub-step numbers (such as '10B') indicate the outcome of the enclosing CHECK (B always indicates that the 'absent' branch was taken). Students found even this simple SOLO-78 trace quite useful. MacSOLO's trace includes parameter-passing details and a single-step facility to further clarify the inner workings of the SOLO machine.

(17) Automatic filing facilities. All procedure definitions and changes to the data base are permanently stored when the user logs out. Upon the next login, everything is exactly the same as when the user last typed 'BYE'. All filing details are taken care of behind the scenes, so the beginner doesn't have to know about 'DISKOUT' or related file-manipulation operations.

5. THE CURRICULUM

Students are given a primer/workbook [10] to introduce them to SOLO. The layout, contents, and sequencing of the primer each contribute to the overall 'SOLO experience'. Many of the strengths of the primer are products of the Open

University production cycle of writing, criticism, re-writing, developmental testing in field conditions, and final polishing. Here are the features of the primer which could be of most benefit to designers of curriculum material to accompany other programming languages:

(1) Clear objectives. The objectives shown earlier in this paper (Section 2) are laid out for the students' benefit in a 'study guide' at the very beginning of the primer. This ensures that students have an idea of what they are expected to achieve, and gives them a way of assessing their own progress on a long-term scale (two weeks, in this case). Progress on individual parts of the booklet is more easily ascertained by means of 'self-assessment questions', with answers provided at the back of the primer.

(2) Highly motivating and obviously relevant 'entry point'. Traditional curriculum sequences involving simple examples such as PRINT "HELLO" or PRINT 3 + 4, followed by discussions of variable assignment, control flow, list structures, etc., are unrewarding for non-Computer Science students. Our students begin by experimenting with assertional data base and pattern matching examples. These are couched in terms of elementary models of human memory, which are directly relevant to the students' main academic interest. It is important for students to be able to do something powerful and exciting during the first session at a computer terminal. Turtle geometry is an example of one

such activity which has been used successfully as an entry point into computing for school children and university students alike [1, 26]. Alternative entry points need to be explored, with an eye towards arousing the interest of students whose primary motivation comes from their dedication to some other discipline. Data base manipulation is one of many possibilities. Others include word processing, music synthesis, games, and laboratory simulation. The virtues of simulation as a motivating focus for computing activities have been amply demonstrated by the work of Kay [18] and Goldberg [13].

(3) Warmth without condescension. The need to make the curriculum material non-threatening had to be balanced against the fact that the students were working adults paying hard-earned money to take an intensive course. Hand-written or cartoon-style presentations, which work well with children learning about computers, were unacceptable to our target population. The solution was to present the material in a totally serious vein, but to provide lots of hand-holding commentary for clarification, and the occasional 'cuddly dog' drawing in the margin for atmosphere. This helped to put the threatened students at ease, and was not regarded as silly or insulting by those who already had some programming experience.

(4) Consistent metaphorical description of virtual machine. Following du Boulay, O'Shea & Monk [8] we chose a single metaphor for describing the working of the SOLO machine (a pool of autonomous 'experts' waiting to be 'activated' and going to work on something like a huge telephone directory). The important point here is the use of a single metaphor and a single technical term for each concept in the primer (e.g. 'slot' is used as a metaphor for 'parameter'). The early parts of the primer provide the students with a concrete model of the underlying transactions occuring within the SOLO machine. As argued in [22], such concreteness can pay large dividends, particularly for 'low-ability' students. Students seem to appreciate the clarity which results from using concrete models and from minimizing the proliferation of metaphors and technical terms.

(5) Use of nonsense words in examples. Beginners often attribute too much intelligence to the machine with which they are working. For instance, students tend to believe that 'WORKSFOR' has some meaning in the triple JOHN---WORKSFOR--->FRED. The primer emphasizes that 'WORKSFOR' only derives its meaning from the procedures which students define, and drives home this point by working through several examples based on nonsense triples, e.g. NOTE FOO---BAZ--->GORT, CHECK FOO---BAZ--->?, etc. Future editions of the primer will rely even more on nonsense-word examples, since some students insist on imparting deep

significance to English-like words in the data base.

(6) Clearly demarcated 'paper and pencil' vs. 'terminal' activities. The fashionable 'try it' philosophy which has dominated primers in recent years (e.g. [2, 34]) is not necessarily the best way to teach all aspects of computing. Clearly, hands-on interactive exercises are very rewarding and are of utmost importance for novices, but it is essential visibly to distinguish between those activities which are meant to be done while sitting at the computer and those activities which can (or should) be done using paper and pencil. Our primer highlights 'study centre (computer terminal) activities' with a wide green stripe along the border of the page.

(7) Delineation of primer and reference manual functions. Many programming instruction manuals fail to distinguish between the role of a primer and that of a reference manual. The former is good for a step-by-step introduction, but useless for finding the quick answer to a specific question, while the latter is handy for looking things up, but not good for self-teaching. The SOLO self-instruction manual is first and foremost a primer. Because of the simplicity of SOLO, the requirements of a reference manual are easily met by having a readily-accessible summary chart of SOLO printed on the back cover of the primer, with the most salient page references printed in bold typeface in a box.

(8) Signposting the student's progress. One of the most difficult aspects of teaching computing to non-computer specialists is maintaining their interest throughout the entire curriculum. They soon get bored by the 'fun' examples used in the introduction, and they don't possess enough expertise to be goaded on by the promise that they will be writing 'recursive inference schemas'. During the developmental testing (pilot) phase of our curriculum design, we found students complaining that they couldn't see why they were doing all these activities. A detailed rationale presented at the beginning was too abstract for students who had no programming background at all, and a presentation at the end (when they could finally understand all of the terminology) was no good because students would give up before getting that far. To circumvent this problem, 'progress boxes' were introduced throughout the primer. These boxes provide a summary of what the students have just done, why they have done it, and where they are headed with the new tools they have just acquired. Early progress boxes motivate the students with promises of future achievements, expressed in terms of the one or two concepts they know, while later progress boxes get increasingly specific, referring to the development of skill in writing searching and inference programs.

6. CONCLUSIONS

SOLO is very simple in conception, and provides a unique combination of attributes which make it ideal as a two-week introduction to computing for non-computer specialists. The special features of the language, environment, and curriculum which have paid high dividends for Open University students are summarized below:

LANGUAGE
* Simple virtual machine
* Consistent use of virtual machine
* Powerful mnemonics
* Not english-like
* Iconic symbols
* Distinct symbols for distinct usage
* Entirely non-numerical
* Small number of primitives
* Uniform syntax
* Encourages top-down programming style
* Forced modularity
* No backward jumps allowed
* Imperative-style conditionals
* Obligatory 'then' and 'else' branches
* Annotated conditional branches
* Obligatory flow of control specification

ENVIRONMENT
* Login trap
* Personalized environment
* Typing lesson for first time users
* Visible workings of inner machine
* Automatic spelling correction of obvious errors
* User confirmation of tricky spelling cases
* Automatic prompting and indenting for both branches of conditional clauses
* Forced explicitness with respect to flow of control
* Inviolability of complex syntax by later editing
* Interpretation of appropriate intentions, given the surrounding context
* Help facilities
* On-line (human) tutorial aid
* Ability to share code with other students
* Ability for users to modify the environment
* Tracing of user-defined procedures
* Automatic filing facilities

CURRICULUM
* Clear objectives
* Highly motivating and obviously relevant entry point
* Warmth without condescension
* Consistent metaphorical description of virtual machine
* Use of nonsense words in examples
* Clearly demarcated paper and pencil vs. terminal activities
* Delineation of primer and reference manual functions
* Signposting of students' progress

One of the main lessons of SOLO is that software designers hoping to have any significant impact on a large population of users must pay a great deal of attention to the interaction of the language, the environment, and the curriculum. A weakness in any of the three can seriously undermine an entire project. Good examples of what can be achieved by this approach are UCSD-Pascal [3], and the Sussex POP11 system [14]. These projects have been successful because the designers took the problems of language, environment, and curriculum quite seriously, and were unwilling to compromise in developing an entire package.

A similar uncompromising attitude has characterized the whole development cycle underlying SOLO. The constraints imposed by the combination of distance-teaching, an audience of Social Science students, the lack of formal prerequisites, and a maximum involvement of two weeks per student led my computer science colleagues in the U.S. and the U.K. to predict that the whole project was doomed from the start. Yet our students acquire demonstrable competence at writing programs which illustrate the concepts mentioned

in Section 1. Part of the secret of this success is the iterative cycle of empirical observation and ensuing refinement of the language, the environment, and the curriculum. Our pilot students could cope with an English-like language, but found that it involved too much typing. The pilot students also suffered from lack of motivation. The motivation problem was tackled by modifying the primer to include examples of obvious relevance to the course at the very beginning, and the language was trimmed down to minimize keystrokes. An in-depth study of SOLO-78 users [12, 21] revealed an overzealous spelling corrector and hard-to-use HELP and TRACE facilities. These in turn were streamlined in later implementations of SOLO.

One of the most difficult issues over the years has been the delicate balance between the principle of consistency, discussed in Section 3, and the need to provide intelligent correction of user errors, so as not to burden the users with needless retyping of silly mistakes. Intelligent error correction leads one to sacrifice consistency in favor of convenience. Although convenience often makes things easier in the short term, it can confuse things in the long term, especially if the student mistakenly attributes cleverness to simple cosmetics. A glaring example of this in SOLO is the use of the word 'TO', which was chosen on cosmetic grounds-- it seemed quite convenient, and LOGO had used it with apparent success for years. Nevertheless, the word 'DEFINE' would have been more

consistent-- in fact du Boulay & O'Shea [7] eliminated the word 'TO' in favor of 'DEFINE' in one dialect of LOGO for precisely this reason. We are now considering doing the same, having observed that students who use 'TO' may begin thinking that all procedures must be verbs [16].

The issue of convenience is not to be totally ignored, however. When a user has typed something which makes his or her intentions obvious, even though the line may contain a minor error, intelligent user-aids should come to the rescue, rather than dogmatically forcing the user to re-type the offending line. Thus, MacSOLO deals with input lines such as:

```
SOLO: FIDO ISA DOG
```

by responding

```
DID YOU INTEND TO TYPE 'NOTE FIDO ISA DOG'?
(YES OR NO):
```

even though the most consistent response, in terms of the inner workings of the SOLO virtual machine, would actually be

```
I DON'T HAVE A PROCEDURE CALLED 'FIDO'
```

As long as the user realizes that an intelligent user-aid is distinct from the SOLO virtual machine, it is in fact possible to achieve both consistency and convenience.

What about the drawbacks of SOLO? Its biggest strength, that it is fanatically tailor-made for a particular type of student working to a particular time scale, turns out to have a serious side-effect: just when

students are happily constructing propositional representations and writing inference programs in SOLO, they begin to get frustrated by its limitations. They want to concatenate strings, perform arithmetic, work with n-tuples rather than triples, and so on. The exciting thing is that they develop many of these notions themselves. The frustrating thing is that SOLO cannot cater to their needs.

An enhancement of the language, though technically feasible, is definitely not in the cards. SOLO was designed to explore the possibility of taking students in very adverse circumstances and handing them a pleasant and moderately powerful computing environment on a silver platter. The purpose of this excercise was to see whether psychology students could quickly learn some elementary computing concepts with a minimum of fuss. Now that this has been shown to be possible, the next logical step would be to apply the lessons learned from SOLO to the development of a two-week introduction to a more widely available language. Given that we are teaching our students elementary AI concepts, an a priori top contender for this honour is LISP. It would be possible to tailor the curriculum to a subset of LISP which shared SOLO's semantics. Students could be introduced to an appropriate set of data base manipulation primitives (perhaps with the same names as those of SOLO), and not told about the lower-level primitives of LISP for the first few weeks. A function keypad would enable students' early interactions

with LISP to be keystroke-for-keystroke identical to their current interactions with Micro-SOLO (a syntax-directed editor would take care of filling in parentheses for them). After two or three weeks, the enforced use of triples (using LISP's GET and PUTPROP) could be lifted, and n-tuples (i.e. lists) could be introduced. Finally, the underlying primitives could be revealed, and a full-blown LISP could be used. The virtue of this approach is that students could have their cake and eat it too-- they could do powerful and exciting things on the first day, using data base assertions and pattern-matching; they could still gain from the benefits of early hand-holding as they do in SOLO, yet have a much more powerful language open up before their eyes at the end of a two or three week introduction.

The above scenario is now under active exploration at the Open University, because the Cognitive Psychology course for which SOLO was developed is being totally remade for presentation in 1986. SOLO has been a very useful testbed: its simple semantics have been ideal for related research projects concerning novices' understanding of recursion [17] and the development of automated debugging aids and programmers' apprentice tools for novices [15, 19, 20]. If nothing else, SOLO has shown that the intensive development of a language, its surrounding environment and curriculum materials, coupled with an active series of empirical investigations, can be of tremendous benefit in developing a practical and successful software product for use by large numbers of computer-naive students.

REFERENCES

1. Abelson, H., & diSessa, A. Turtle geometry: the computer as a medium for exploring mathematics. MIT Press, Cambridge, Mass., 1981.

2. Alcock, D. Illustrating BASIC. Cambridge University Press, Cambridge, 1977.

3. Bowles, K. Microcomputer problem solving using Pascal. Springer, New York, 1977.

4. Bramer, M. Using computers in distance education: the first ten years of the British Open University. Technical report no. 1, Computer Assisted Learning Research Group, The Open University, Milton Keynes, England, 1980.

5. Brinch Hansen, P. The architecture of concurrent programs. Prentice-Hall, Englewood Cliffs, N.J., 1977.

6. Brown, J.S., & Rubinstein, R. Recursive functional programming for the student in the humanities and social sciences. Technical report no. 27a, Department of Information and Computer Science, University of California at Irvine, Irvine, Calif., 1974.

7. du Boulay, J.B.H., & O'Shea, T. How to work the LOGO machine: a primer for ELOGO. Edinburgh: Occasional paper no. 4, Department of Artificial Intelligence, University of Edinburgh, 1976.

8. du Boulay, J.B.H., O'Shea, T., & Monk, J. The black box inside the glass box: presenting computing concepts to novices. International Journal of Man Machine Studies, 14, 3 (Apr. 1981) 237-249.

9. Dwyer, T., & Critchfield, M. Multi-computer systems for the support of inventive learning. In P.R. Smith (Ed.), Computer assisted learning: selected papers from the CAL-81 symposium. Pergammon Press, Oxford, 1981.

10. Eisenstadt, M. The SOLO primer. Units 3/4 of Cognitive psychology: a third level course. Open University Press, Milton Keynes, England, 1978.

11. Eisenstadt, M., & Gawronski, A. Micro-SOLO: a tool for elementary AI programming. Technical report no. 29, Computer Assisted Learning Research Group, The Open University, Milton Keynes, England, 1982.

12. Eisenstadt, M., & Lewis, M. The behaviour of novice programmers in a friendly software environment. Technical report no. 4, Human Cognition Research Laboratory, The Open University, Milton Keynes, England, 1982.

13. Goldberg, A. Introducing the Smalltalk-80 system. Byte, 6, 8 (Aug. 1981), 14-26.

14. Hardy, S. POP11: an AI programming environment. In T. O'Shea & M. Eisenstadt (Eds.), Artificial intelligence skills. Harper & Row, New York, in press.

15. Hasemer, T. MacSOLO. Technical report no. 24, Computer Assisted Learning Research Group, The Open University, Milton Keynes, England, 1982.

16. Jones, A. The role of metaphors in novices learning programming. Proceedings of the Fourth Annual Cognitive Science Society Conference, Ann Arbor, Michigan, 1982

17. Kahney, J.H. Problem solving by novice programmers. In T.R.G. Green, S.J. Payne & G.C. van der Veer (Eds.), The psychology of computer use: a European perspective. Academic Press, London, in press.

18. Kay, A. Microelectronics and the personal computer. Scientific American, 237, 3 (Sept. 1977), 230-244.

19. Laubsch, J. & Eisenstadt, M. Domain specific debugging aids for novice programmers. Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-81), Vancouver, B.C. Canada, 1981.

20. Laubsch, J., & Eisenstadt, M. Using temporal abstraction to understand recursive programs involving side effects. Proceedings of the National Conference on Artificial Intelligence (AAAI-82), Pittsburgh, Penn., 1982.

21. Lewis, M. Improving SOLO's user-interface: an empirical study of user behaviour and proposals for cost-effective enhancements to SOLO. Technical report no. 7, Computer Assisted Learning Research Group, The Open University, Milton Keynes, England, 1980.

22. Mayer, R.E. A psychology of learning BASIC. Communications of the ACM, 22, 11 (Nov. 1979), 589-593.

23. McDermott, D.V., & Sussman, G.J. The CONNIVER reference manual. Cambridge, Mass., Memo AIM-259A, Artificial Intelligence Laboratory, Massachussetts Institute of Technology, 1974.

24. Model, M. Monitoring system behavior in a complex computational environment. Technical report no. CS-79-701, Department of Computer Science, Stanford University, Stanford, California, 1979.

25. Norman, D.A., Rumelhart, D.E., & The LNR Research Group. Explorations in cognition. W.H. Freeman, San Francisco, 1975.

26. Papert, S. Mindstorms: children, computers, and powerful ideas. Harvester Press, Brighton, England, 1980.

27. Perry, W. Open University. The Open University Press, Milton Keynes, England, 1976.

28. Ruth, G.R. Intelligent program analysis. Artificial Intelligence, 7, 1 (Jan. 1976), 65-85.

29. Sime, M.E., Green, T.R.G., Guest, D.J. Psychological evaluation of two conditional constructions used in computer languages. International Journal of Man-Machine Studies, 5, (1973), 123-143.

30. Soloway, E., Woolf, B., Rubin, E., & Barth, P. Meno-II: an intelligent tutoring system for novice programmers. Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-81), Vancouver, B.C. Canada, 1981.

31. Teitelbaum, T. & Reps, T. The Cornell program synthesizer: a syntax-directed programming environment. Communications of the ACM, 24, 9 (Sept. 1981), 563-573.

32. Teitelman, W. Interlisp reference manual. Xerox Palo Alto Research Center, Palo Alto, Calif., 1978.

33. Waters, R.C. The programmer's apprentice: knowledge based program editing. IEEE Transactions on Software Engineering, 8, 1 (Jan. 1982), 1-12.

34. Winston, P.H., & Horn, B.K.P. LISP. Addison-Wesley, Reading, Mass., 1981.